Mechanizing Game-Based Proofs of Security Protocols

Bruno Blanchet

INRIA, École Normale Supérieure, CNRS, Paris
blanchet@di.ens.fr

August 2011

Preliminary information

- Exercises available on my home page:

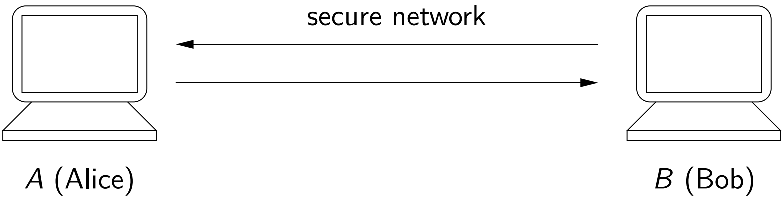
  `http://www.di.ens.fr/~blanchet`

- For the last two exercises, installing CryptoVerif on your notebook would be useful.
  Can be downloaded from my home page.

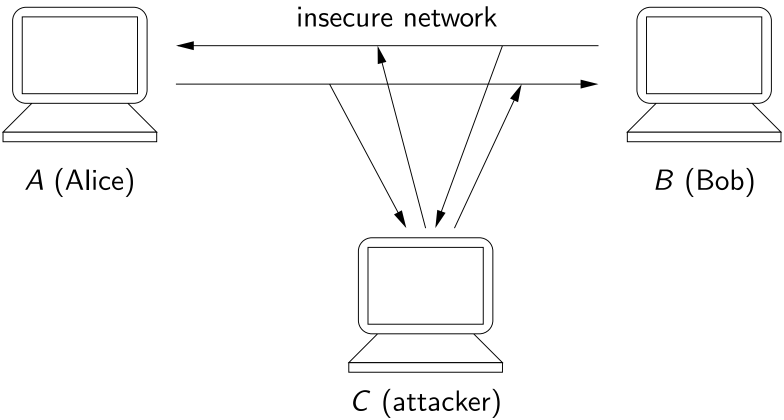
Outline

1. Introduction: verification of security protocols in the computational model
2. Using CryptoVerif
3. Proof technique: game transformations, proof strategy
4. Two examples:
   - Encrypt-then-MAC
   - FDH
5. Conclusion, future directions

Communications over a secure network



Communications over an insecure network



A talks to B on an insecure network

$\Rightarrow$ need for cryptography in order to make communications secure
for instance, encrypt messages to preserve secrets.

Some cryptographic primitives

Cryptographic primitives

Algorithms that are frequently used to build computer security systems. These routines include, but are not limited to, encryption and signature functions.

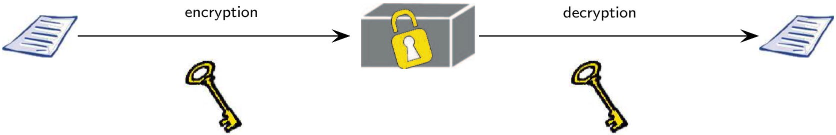
based on slides by Stéphanie Delaune

Some cryptographic primitives

Cryptographic primitives

Algorithms that are frequently used to build computer security systems. These routines include, but are not limited to, encryption and signature functions.

Symmetric encryption



$\longrightarrow$ Examples: Caesar encryption, DES, AES, ...

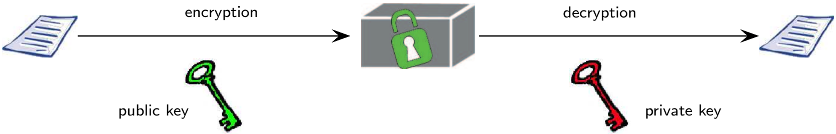
based on slides by Stéphanie Delaune

Some cryptographic primitives

Cryptographic primitives

Algorithms that are frequently used to build computer security systems. These routines include, but are not limited to, encryption and signature functions.

Asymmetric encryption



⟶ Examples: RSA, El Gamal, . . .

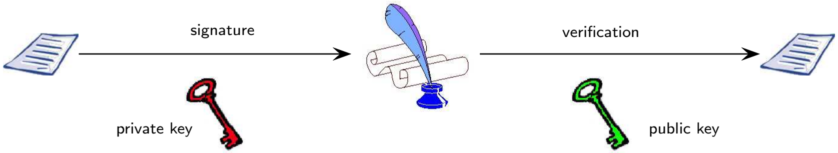
based on slides by Stéphanie Delaune

Some cryptographic primitives

Cryptographic primitives

Algorithms that are frequently used to build computer security systems. These routines include, but are not limited to, encryption and signature functions.

Signature



$\longrightarrow$ Examples: RSA, ...

based on slides by Stéphanie Delaune

Examples

Many protocols exist, for various goals:

- secure channels: SSH (Secure SHell);
  SSL (Secure Socket Layer), renamed TLS (Transport Layer Security);
  IPsec
- e-voting
- contract signing
- certified email
- wifi (WEP/WPA/WPA2)
- banking
- mobile phones
- ...

Why verify security protocols ?

The verification of security protocols has been and is still a very active research area.

- Their design is error prone.
- Security errors are not detected by testing:
  they appear only in the presence of an adversary.
- Errors can have serious consequences.

Models of protocols

Active attacker:

- the attacker can intercept all messages sent on the network
- he can compute messages
- he can send messages on the network

Models of protocols: the formal model

The formal model or "Dolev-Yao model" is due to Needham and Schroeder [1978] and Dolev and Yao [1983].

- The cryptographic primitives are blackboxes.
- The messages are terms on these primitives.
- The attacker is restricted to compute only using these primitives.
  $\Rightarrow$ perfect cryptography assumption

One can add equations between primitives, but in any case, one makes the hypothesis that the only equalities are those given by these equations.

This model makes automatic proofs relatively easy (AVISPA, ProVerif, ...).

Models of protocols: the computational model

The computational model has been developed at the beginning of the 1980's by Goldwasser, Micali, Rivest, Yao, and others.

- The messages are bitstrings.
- The cryptographic primitives are functions on bitstrings.
- The attacker is any probabilistic (polynomial-time) Turing machine.

This model is much more realistic than the formal model, but until recently proofs were only manual.

Models of protocols: side channels

The computational model is still just a model, which does not exactly match reality.

In particular, it ignores side channels:

- timing
- power consumption
- noise
- physical attacks against smart cards

which can give additional information.

We will still focus on the computational model.

Obtaining proofs in the computational model

Two approaches for the automatic proof of cryptographic protocols in a computational model:

- Indirect approach:

  1) Make a Dolev-Yao proof.
  2) Use a theorem that shows the soundness of the Dolev-Yao approach with respect to the computational model.

  Approach pioneered by Abadi&Rogaway [2000]; many works since then.

- Direct approach:

  Design automatic tools for proving protocols in the computational model.

  Approach pioneered by Laud [2004].

Direct versus indirect approach

The indirect approach allows more reuse of previous work, but it has limitations:

- Hypotheses have to be added to make sure that the computational and Dolev-Yao models coincide.
- The allowed cryptographic primitives are often limited, and only ideal, not very practical primitives can be used.
- Using the Dolev-Yao model is actually a (big) detour;
  The computational definitions of primitives fit the computational security properties to prove.
  They do not fit the Dolev-Yao model.

We decided to focus on the direct approach.

Proofs by sequences of games

Proofs in the computational model are typically proofs by sequences of games [Shoup, Bellare&Rogaway]:

- The first game is the real protocol.
- One goes from one game to the next by syntactic transformations or by applying the definition of security of a cryptographic primitive. The difference of probability between consecutive games is negligible.
- The last game is "ideal": the security property is obvious from the form of the game. (The advantage of the adversary is 0 for this game.)

Game 0: Protocol to prove $\longleftrightarrow$ (p_1 negligible) **Game 1** $\longleftrightarrow$ (p_2 negligible) $\cdots$ $\longleftrightarrow$ (p_n negligible) **Game n**: Property obvious

Mechanizing proofs by sequences of games

- CryptoVerif
  - Will be the main topic of this course
  - A similar tool has been built by Tšahhirov and Laud [2007], using a different game representation (dependency graph).
- CertiCrypt
- F7 and typing

Mechanizing proofs by sequences of games

- CryptoVerif
- CertiCrypt, see `http://software.imdea.org/~szanella/`
  - Machine-checked cryptographic proofs in Coq
  - Interesting case studies, *e.g.* OAEP
  - Good for proving primitives: can prove complex mathematical theorems
  - Requires a lot of human effort
  - Improved by EasyCrypt: generates CertiCrypt proofs from proof sketches (sequence of games and hints)
- F7 and typing

Mechanizing proofs by sequences of games

- CryptoVerif
- CertiCrypt
- F7 and typing, see Fournet et al
  - Use a type system to determine whether a game transformation can be applied.
    If yes, apply the game transformation, and repeat.
  - Allows the verification of implementations of protocols.

Another typing approach is computationally-sound type systems:
if the protocol is well-typed, it is secure in the computational model.

CryptoVerif, http://www.cryptoverif.ens.fr/

CryptoVerif is an automatic prover that:

- generates proofs by sequences of games.
- proves secrecy and correspondence properties.
- provides a generic method for specifying properties of cryptographic primitives which handles MACs (message authentication codes), symmetric encryption, public-key encryption, signatures, hash functions, Diffie-Hellman key agreements, . . .
- works for N sessions (polynomial in the security parameter), with an active adversary.
- gives a bound on the probability of an attack (exact security).

Input and output of the tool

1. Prepare the input file containing
   - the specification of the protocol to study (initial game),
   - the security assumptions on the cryptographic primitives,
   - the security properties to prove.
2. Run CryptoVerif
3. CryptoVerif outputs
   - the sequence of games that leads to the proof,
   - a succinct explanation of the transformations performed between games,
   - an upper bound of the probability of success of an attack.

Process calculus for games

Games are formalized in a process calculus:

- It is adapted from the pi calculus.
- The semantics is purely probabilistic (no non-determinism).
- The runtime of processes is bounded:
  - bounded number of copies of processes,
  - bounded length of messages on channels.
- Extension to arrays.

Process calculus for games: terms

Terms represent computations on messages (bitstrings).

$M ::=$	terms
$\quad x, y, z, x[M_1, \dots, M_n]$	variable
$\quad f(M_1, \dots, M_n)$	function application

Function symbols f correspond to functions computable by deterministic Turing machines that always terminate.

Process calculus for games: processes

$Q ::=$	input process
0	end
$Q \mid Q'$	parallel composition
$!^{i \leq N} Q$	replication N times
$\mathbf{newChannel}\ c; Q$	restriction for channels
$c(x_1 : T_1, \ldots, x_m : T_m); P$	input
$P ::=$	output process
$\mathbf{yield}$	end
$\overline{c}\langle M_1, \ldots, M_m\rangle; Q$	output
$\mathbf{event}\ e(M_1, \ldots, M_m); P$	event
$\mathbf{new}\ x : T; P$	random number generation (uniform)
$\mathbf{let}\ x : T = M\ \mathbf{in}\ P$	assignment
$\mathbf{if}\ M\ \mathbf{then}\ P\ \mathbf{else}\ P'$	conditional
$\mathbf{find}\ j \leq N\ \mathbf{suchthat}\ \mathbf{defined}(x[j], \ldots) \wedge M\ \mathbf{then}\ P\ \mathbf{else}\ P'$	array lookup

Example: 1. symmetric encryption

We consider a probabilistic, length-revealing encryption scheme.

Definition (Symmetric encryption scheme SE)

- (Randomized) key generation function $kgen$.
- (Randomized) encryption function $enc(m, k, r')$ takes as input a message m, a key k, and random coins r'.
- Decryption function $dec(c, k)$ such that

$$dec(enc(m, kgen(r), r'), kgen(r)) = i_\perp(m)$$

The decryption returns a bitstring or $\perp$:

- $\perp$ when decryption fails,
- the cleartext when decryption succeeds.

The injection $i_\perp$ maps a bitstring to the same bitstring in bitstring $\cup \{\perp\}$.

Example: 2. MAC

Definition (Message Authentication Code scheme MAC)

- (Randomized) key generation function $mkgen$.
- MAC function $mac(m, k)$ takes as input a message m and a key k.
- Verification function $verify(m, k, t)$ such that

$$verify(m, k, mac(m, k)) = true.$$

A MAC is essentially a keyed hash function.

A MAC guarantees the integrity and authenticity of the message because only someone who knows the secret key can build the MAC.

Example: 3. encrypt-then-MAC

We define an authenticated encryption scheme by the encrypt-then-MAC construction:

$$enc'(m, (k, mk), r'') = e, mac(e, mk) \text{ where } e = enc(m, k, r'').$$

A basic example of protocol using encrypt-then-MAC:

- A and B initially share an encryption key k and a MAC key mk.
- A sends to B a fresh key k' encrypted under authenticated encryption, implemented as encrypt-then-MAC.

$$A \rightarrow B : e = enc(k', k, r''), mac(e, mk) \qquad k' \text{ fresh}$$

k' should remain secret.

Example: initialization

$$A \to B : e = enc(k', k, r''), mac(e, mk) \qquad k' \text{ fresh}$$

$$\begin{aligned} Q_0 = {} & start(); \textbf{new } r : keyseed; \textbf{let } k : key = kgen(r) \textbf{ in} \\ & \textbf{new } r' : mkeyseed; \textbf{let } mk : mkey = mkgen(r') \textbf{ in } \overline{c}\langle\rangle; (Q_A \mid Q_B) \end{aligned}$$

Initialization of keys:

1. The process Q_0 waits for a message on channel *start* to start running.
   The adversary triggers this process.
2. Q_0 generates encryption and MAC keys, k and mk respectively, using the key generation algorithms *kgen* and *mkgen*.
3. Q_0 returns control to the adversary by the output $\overline{c}\langle\rangle$.
   Q_A and Q_B represent the actions of A and B (see next slides).

Example: role of A

$$A \to B : e = enc(k', k, r''), mac(e, mk) \qquad k' \text{ fresh}$$

$$\begin{aligned} Q_A = {} & !^{i \leq n} c_A(); \textbf{new } k' : key; \textbf{new } r'' : coins; \\ & \textbf{let } e : bitstring = enc(k2b(k'), k, r'') \textbf{ in} \\ & \overline{c_A}\langle e, mac(e, mk)\rangle \end{aligned}$$

Role of A:

1. $!^{i \leq n}$ represents n copies, indexed by $i \in [1, n]$
   The protocol can be run n times (polynomial in the security parameter).
2. The process is triggered when a message is sent on c_A by the adversary.
3. The process chooses a fresh key k' and sends the message on channel c_A.

Example: role of B

$$A \to B : e = enc(k', k, r''), mac(e, mk) \qquad k' \text{ fresh}$$

$$\begin{aligned} Q_B = {} & !^{i' \leq n} c_B(e' : bitstring, ma : macstring); \\ & \textbf{if } verify(e', mk, ma) \textbf{ then} \\ & \textbf{let } i_\perp(k2b(k'')) = dec(e', k) \textbf{ in } \overline{c_B}\langle\rangle \end{aligned}$$

Role of B:

1. n copies, as for Q_A.
2. The process Q_B waits for the message on channel c_B.
3. It verifies the MAC, decrypts, and stores the key in k''.

Example: summary of the initial game

$$A \rightarrow B : e = enc(k', k, r''), mac(e, mk) \qquad k' \text{ fresh}$$

$$\begin{aligned}
Q_0 = {} & start(); \textbf{new}\ r : keyseed; \textbf{let}\ k : key = kgen(r)\ \textbf{in} \\
& \textbf{new}\ r' : mkeyseed; \textbf{let}\ mk : mkey = mkgen(r')\ \textbf{in}\ \overline{c}\langle\rangle; (Q_A \mid Q_B)
\end{aligned}$$

$$\begin{aligned}
Q_A = {} & !^{i \leq n} c_A(); \textbf{new}\ k' : key; \textbf{new}\ r'' : coins; \\
& \textbf{let}\ e : bitstring = enc(k2b(k'), k, r'')\ \textbf{in} \\
& \overline{c_A}\langle e, mac(e, mk)\rangle
\end{aligned}$$

$$\begin{aligned}
Q_B = {} & !^{i' \leq n} c_B(e' : bitstring, ma : macstring); \\
& \textbf{if}\ verify(e', mk, ma)\ \textbf{then} \\
& \textbf{let}\ i_\perp(k2b(k'')) = dec(e', k)\ \textbf{in}\ \overline{c_B}\langle\rangle
\end{aligned}$$

Security assumptions on primitives

The most frequent cryptographic primitives are already specified in a library. The user can use them without redefining them.

In the example:

- The MAC is UF-CMA (unforgeable under chosen message attacks). An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).

Security assumptions on primitives

The most frequent cryptographic primitives are already specified in a library. The user can use them without redefining them.

In the example:

- The MAC is UF-CMA (unforgeable under chosen message attacks).
  An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).
- The encryption is IND-CPA (indistinguishable under chosen plaintext attacks).
  An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.

Security assumptions on primitives

The most frequent cryptographic primitives are already specified in a library. The user can use them without redefining them.

In the example:

- The MAC is UF-CMA (unforgeable under chosen message attacks).
  An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).
- The encryption is IND-CPA (indistinguishable under chosen plaintext attacks).
  An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.
- All keys have the same length: **forall** $y : key; Z(k2b(y)) = Z_k$.

Security properties to prove

In the example:

- One-session secrecy of k'': each k'' is indistinguishable from a random number.
- Secrecy of k'': the k'' are indistinguishable from independent random numbers.

Demo

- CryptoVerif input file: enc-then-MAC.cv
- library of primitives
- run CryptoVerif
- output

Arrays

Arrays replace lists often used in cryptographic proofs.

They avoid the need for explicit list insertion instructions, which would be hard to guess for an automatic tool.

A variable defined under a replication is implicitly an array:

$$\begin{aligned} Q_A = {}& !^{i \leq n} c_A(); \textbf{new } k'[i] : \mathit{key}; \textbf{new } r''[i] : \mathit{coins}; \\ & \textbf{let } e[i] : \mathit{bitstring} = \mathit{enc}(k2b(k'[i]), k, r''[i]) \textbf{ in} \\ & \overline{c_A}\langle e[i], \mathit{mac}(e[i], mk)\rangle \end{aligned}$$

Requirements:

- Only variables with the current indices can be assigned.
- Variables may be defined at several places, but only one definition can be executed for the same indices.
  (**if** ... **then let** $x = M$ **in** P **else let** $x = M'$ **in** P' is ok)

So each array cell can be assigned at most once.

Arrays (continued)

find performs an array lookup:

$$!^{i \leq N} \ldots \textbf{let } x = M \textbf{ in } P$$
$$\mid\, !^{i' \leq N'} c(y : T)\textbf{find } j \leq N \textbf{ suchthat defined}(x[j]) \wedge y = x[j] \textbf{ then } \ldots$$

Note that **find** is here used outside the scope of x.

This is the only way of getting access to values of variables in other sessions.

When several array elements satisfy the condition of the **find**, the returned index is chosen randomly, with uniform probability.

Indistinguishability as observational equivalence

Two processes (games) Q_1, Q_2 are observationally equivalent when the adversary has a negligible probability of distinguishing them: $Q_1 \approx Q_2$.

- The adversary is represented by an acceptable evaluation context $C ::= [\,] \quad C \mid Q \quad Q \mid C \quad \textbf{newChannel } c; C$.
- $C[Q]$ may execute events, collected in a sequence $\mathcal{E}$.
- A distinguisher D takes as input $\mathcal{E}$ and returns **true** or **false**.
    - Example: $D(\mathcal{E}) = \textbf{true}$ if and only if $e \in \mathcal{E}$.
- $\Pr[C[Q] \rightsquigarrow D]$ is the probability that $C[Q]$ executes $\mathcal{E}$ such that $D(\mathcal{E}) = \textbf{true}$.

Definition (Indistinguishability)

We write $Q \approx_p^V Q'$ when, for all evaluation contexts C acceptable for Q and Q' with public variables V and all distinguishers D,

$$|\Pr[C[Q] \rightsquigarrow D] - \Pr[C[Q'] \rightsquigarrow D]| \leq p(C, D).$$

Indistinguishability as observational equivalence

Lemma

1. *Reflexivity:* $Q \approx_0^V Q$.
2. *Symmetry:* $\approx_p^V$ *is symmetric.*
3. *Transitivity: if* $Q \approx_p^V Q'$ *and* $Q' \approx_{p'}^V Q''$, *then* $Q \approx_{p+p'}^V Q''$.
4. *Application of context: if* $Q \approx_p^V Q'$ *and* C *is an evaluation context acceptable for* Q *and* Q' *with public variables* V, *then* $C[Q] \approx_{p'}^{V'} C[Q']$, *where* $p'(C', D) = p(C'[C[\,]], D)$ *and* $V' \subseteq V \cup \mathrm{var}(C)$.

Proof technique

We transform a game G_0 into an observationally equivalent one using:

- observational equivalences $L \approx_p R$ given as axioms and that come from security assumptions on primitives. These equivalences are used inside a context:
$$G_1 \approx_0 C[L] \approx_{p'} C[R] \approx_0 G_2$$
- syntactic transformations: simplification, expansion of assignments, ...

We obtain a sequence of games $G_0 \approx_{p_1} G_1 \approx \ldots \approx_{p_m} G_m$, which implies $G_0 \approx_{p_1+\cdots+p_m} G_m$.

If some trace property holds up to probability p in G_m, then it holds up to probability $p + p_1 + \cdots + p_m$ in G_0.

MAC: definition of security (UF-CMA)

A MAC guarantees the integrity and authenticity of the message because only someone who knows the secret key can build the MAC.
More formally, $\mathrm{Succ}^{\mathrm{uf-cma}}_{\mathrm{MAC}}(t, q_m, q_v, l)$ is negligible if t is polynomial in the security parameter:

Definition (UnForgeability under Chosen Message Attacks, UF-CMA)

$$\mathrm{Succ}^{\mathrm{uf-cma}}_{\mathrm{MAC}}(t, q_m, q_v, l) =$$

$$\max_{\mathcal{A}} \Pr\left[\begin{array}{l} k \stackrel{R}{\leftarrow} \mathit{mkgen}; (m, t) \leftarrow \mathcal{A}^{\mathit{mac}(.,k),\mathit{verify}(.,k,.)} : \mathit{verify}(m, k, t) \wedge \\ m \text{ was never queried to the oracle } \mathit{mac}(., k) \end{array}\right]$$

where $\mathcal{A}$ runs in time at most t,
calls $\mathit{mac}(., k)$ at most q_m times with messages of length at most l,
calls $\mathit{verify}(., k, .)$ at most q_v times with messages of length at most l.

MAC: intuition behind the CryptoVerif definition

By the previous definition, up to negligible probability,

- the adversary cannot forge a correct MAC
- so, assuming $k \stackrel{R}{\leftarrow} mkgen$ is used only for generating and verifying MACs, the verification of a MAC with $verify(m, k, t)$ can succeed only if m is in the list (array) of messages whose $mac(\cdot, k)$ has been computed by the protocol
- so we can replace a call to *verify* with an array lookup:
  if the call to *mac* is $mac(x, k)$, we replace $verify(m, k, t)$ with

$$\textbf{find } j \leq N \textbf{ suchthat defined}(x[j]) \wedge (m = x[j]) \wedge verify(m, k, t) \textbf{ then true else false}$$

MAC: CryptoVerif definition

$$verify(m, mkgen(r), mac(m, mkgen(r))) = \mathbf{true}$$

$$!^{N''}\mathbf{new}\ r : mkeyseed; ($$
$$\quad !^{N} Omac(x : bitstring) := mac(x, mkgen(r)),$$
$$\quad !^{N'} Overify(m : bitstring, t : macstring) := verify(m, mkgen(r), t))$$
$$\approx$$
$$!^{N''}\mathbf{new}\ r : mkeyseed; ($$
$$\quad !^{N} Omac(x : bitstring) := mac(x, mkgen(r)),$$
$$\quad !^{N'} Overify(m : bitstring, t : macstring) :=$$
$$\qquad \mathbf{find}\ j \leq N\ \mathbf{suchthat}\ \mathbf{defined}(x[j]) \wedge (m = x[j]) \wedge$$
$$\qquad\qquad verify(m, mkgen(r), t)\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \mathbf{false})$$

MAC: CryptoVerif definition

$$\mathit{verify}(m, \mathit{mkgen}(r), \mathit{mac}(m, \mathit{mkgen}(r))) = \mathbf{true}$$

$$\begin{array}{l}
!^{N''} \mathbf{new}\ r : \mathit{mkeyseed}; (\\
\quad !^{N} \mathit{Omac}(x : \mathit{bitstring}) := \mathit{mac}(x, \mathit{mkgen}(r)),\\
\quad !^{N'} \mathit{Overify}(m : \mathit{bitstring}, t : \mathit{macstring}) := \mathit{verify}(m, \mathit{mkgen}(r), t))
\end{array}$$

$$\approx_{\substack{N'' \times \mathrm{Succ}^{\mathrm{uf-cma}}_{\mathrm{MAC}}(\mathbf{time}+(N''-1)(\mathbf{time}(\mathit{mkgen})+N\,\mathbf{time}(\mathit{mac},\mathrm{maxl}(x))+\\ N'\,\mathbf{time}(\mathit{verify},\mathrm{maxl}(m)),N,N',\max(\mathrm{maxl}(x),\mathrm{maxl}(m)))}}$$

$$\begin{array}{l}
!^{N''} \mathbf{new}\ r : \mathit{mkeyseed}; (\\
\quad !^{N} \mathit{Omac}(x : \mathit{bitstring}) := \mathit{mac}'(x, \mathit{mkgen}'(r)),\\
\quad !^{N'} \mathit{Overify}(m : \mathit{bitstring}, t : \mathit{macstring}) :=\\
\qquad \mathbf{find}\ j \leq N\ \mathbf{suchthat}\ \mathbf{defined}(x[j]) \wedge (m = x[j]) \wedge\\
\qquad\qquad \mathit{verify}'(m, \mathit{mkgen}'(r), t)\ \mathbf{then\ true\ else\ false})
\end{array}$$

CryptoVerif understands such specifications of primitives.

Exercise 1

The advantage of the adversary against strong unforgeability under chosen message attacks (SUF-CMA) of MACs is:

$$\mathrm{Succ}^{\mathrm{suf-cma}}_{\mathrm{MAC}}(t, q_m, q_v, l) =$$
$$\max_{\mathcal{A}} \Pr\left[\begin{array}{l} k \stackrel{R}{\leftarrow} \mathit{mkgen}; (m,t) \leftarrow \mathcal{A}^{\mathit{mac}(.,k),\mathit{verify}(.,k,.)} : \mathit{verify}(m,k,t) \wedge \\ t \text{ is not the result of calling the oracle } \mathit{mac}(.,k) \text{ on } m \end{array}\right]$$

where $\mathcal{A}$ runs in time at most t,
calls $\mathit{mac}(.,k)$ at most q_m times with messages of length at most l,
calls $\mathit{verify}(.,k,.)$ at most q_v times with messages of length at most l.

Represent SUF-CMA MACs in the CryptoVerif formalism.

Exercise 2

A signature scheme S consists of

- a key generation algorithm $(pk, sk) \stackrel{R}{\leftarrow} kgen$
- a signature algorithm $sign(m, sk)$
- a verification algorithm $verify(m, pk, s)$

such that $verify(m, pk, sign(m, sk)) = 1$.

The advantage of the adversary against unforgeability under chosen message attacks (UF-CMA) of signatures is:

$$\mathrm{Succ}_{\mathrm{S}}^{\mathrm{uf-cma}}(t, q_s, l) =$$

$$\max_{\mathcal{A}} \Pr\left[\begin{array}{l}(pk, sk) \stackrel{R}{\leftarrow} kgen; (m, s) \leftarrow \mathcal{A}^{sign(.,sk)}(pk) : verify(m, pk, s) \wedge \\ m \text{ was never queried to the oracle } sign(., sk)\end{array}\right]$$

where $\mathcal{A}$ runs in time at most t,
calls $sign(., sk)$ at most q_s times with messages of length at most l.
Represent UF-CMA signatures in the CryptoVerif formalism.

MAC: using the CryptoVerif definition

CryptoVerif applies the previous rule automatically in any context, perhaps containing several occurrences of *mac* and of *verify*:

- Each occurrence of *mac* is replaced with *mac′*.
- Each occurrence of *verify* is replaced with a **find** that looks in all arrays of computed MACs (one array for each occurrence of function *mac*).

Symmetric encryption: definition of security (IND-CPA)

An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.

Definition (INDistinguishability under Chosen Plaintext Attacks, IND-CPA)

$$\mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(t, q_e, l) =$$
$$\max_{\mathcal{A}} 2 \Pr\left[b \stackrel{R}{\leftarrow} \{0,1\}; k \stackrel{R}{\leftarrow} kgen; b' \leftarrow \mathcal{A}^{enc(LR(.,.,b),k)} : b' = b\right] - 1$$

where $\mathcal{A}$ runs in time at most t,
calls $enc(LR(.,.,b), k)$ at most q_e times on messages of length at most l,
$LR(x, y, 0) = x$, $LR(x, y, 1) = y$, and $LR(x, y, b)$ is defined only when x and y have the same length.

Symmetric encryption: CryptoVerif definition

$$dec(enc(m, kgen(r), r'), kgen(r)) = i_\perp(m)$$

$$!^{N'} \mathbf{new}\ r : keyseed; !^{N} Oenc(x : bitstring) :=$$
$$\mathbf{new}\ r' : coins; enc(x, kgen(r), r')$$
$$\approx_{N' \times \mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(\mathbf{time} + (N'-1)(\mathbf{time}(kgen) + N\,\mathbf{time}(enc, \mathrm{maxl}(x)) + N\,\mathbf{time}(Z, \mathrm{maxl}(x))), N, \mathrm{maxl}(x))}$$
$$!^{N'} \mathbf{new}\ r : keyseed; !^{N} Oenc(x : bitstring) :=$$
$$\mathbf{new}\ r' : coins; enc'(Z(x), kgen'(r), r')$$

$Z(x)$ is the bitstring of the same length as x containing only zeroes (for all $x : nonce$, $Z(x) = Znonce$, ...).

Exercise 3

The advantage of the adversary against ciphertext integrity (INT-CTXT) of a symmetric encryption scheme SE is:

$$\mathrm{Succ}_{\mathrm{SE}}^{\mathrm{int-ctxt}}(t, q_e, q_d, l_e, l_d) =$$
$$\max_{\mathcal{A}} \Pr\left[\begin{array}{l} k \stackrel{R}{\leftarrow} kgen; c \leftarrow \mathcal{A}^{enc(.,k), dec(.,k) \neq \perp} : dec(c,k) \neq \perp \wedge \\ c \text{ is not the result of a call to the } enc(.,k) \text{ oracle} \end{array}\right]$$

where $\mathcal{A}$ runs in time at most t,
calls $enc(.,k)$ at most q_e times with messages of length at most l_e,
calls $dec(.,k) \neq \perp$ at most q_d times with messages of length at most l_d.

Represent INT-CTXT encryption in the CryptoVerif formalism.

Exercise 4

A public-key encryption scheme AE consists of

- a key generation algorithm $(pk, sk) \stackrel{R}{\leftarrow} kgen$
- a probabilistic encryption algorithm $enc(m, pk)$
- a decryption algorithm $dec(m, sk)$

such that $dec(enc(m, pk), sk) = m$.

The advantage of the adversary against indistinguishability under adaptive chosen-ciphertext attacks (IND-CCA2) is

$$\mathrm{Succ}_{\mathrm{AE}}^{\mathrm{ind-cca2}}(t, q_d) =$$
$$\max_{\mathcal{A}} 2\Pr\left[\begin{array}{l} b \stackrel{R}{\leftarrow} \{0,1\}; (pk, sk) \stackrel{R}{\leftarrow} kgen; \\ (m_0, m_1, s) \leftarrow \mathcal{A}_1^{dec(.,sk)}(pk); y \leftarrow enc(m_b, pk); \\ b' \leftarrow \mathcal{A}_2^{dec(.,sk)}(m_0, m_1, s, y) : b' = b \wedge \\ \mathcal{A}_2 \text{ has not called } dec(., sk) \text{ on } y \end{array}\right] - 1$$

where $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ runs in time at most t and calls $dec(., sk)$ at most q_d times. Represent IND-CCA2 encryption in the CryptoVerif formalism.

Syntactic transformations (1)

Expansion of assignments: replacing a variable with its value.
(Not completely trivial because of array references.)

Example

If mk is defined by

$$\textbf{let } mk = mkgen(r')$$

and there are no array references to mk, then mk is replaced with $mkgen(r')$ in the game and the definition of mk is removed.

Syntactic transformations (2)

Single assignment renaming: when a variable is assigned at several places, rename it with a distinct name for each assignment.
(Not completely trivial because of array references.)

Example

$$\begin{aligned}
&\mathit{start}();\mathbf{new}\ r_A : T_r;\mathbf{let}\ k_A = \mathit{kgen}(r_A)\ \mathbf{in}\\
&\qquad\mathbf{new}\ r_B : T_r;\mathbf{let}\ k_B = \mathit{kgen}(r_B)\ \mathbf{in}\ \overline{c}\langle\rangle;(Q_K \mid Q_S)\\
&Q_K = !^{i\leq n}c(h : T_h, k : T_k)\\
&\qquad\mathbf{if}\ h = A\ \mathbf{then}\ \mathbf{let}\ k' = k_A\ \mathbf{else}\\
&\qquad\mathbf{if}\ h = B\ \mathbf{then}\ \mathbf{let}\ k' = k_B\ \mathbf{else}\ \mathbf{let}\ k' = k\\
&Q_S = !^{i'\leq n'}c'(h' : T_h);\\
&\qquad\mathbf{find}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(h[j], k'[j]) \wedge h' = h[j]\ \mathbf{then}\ P_1(k'[j])\\
&\qquad\mathbf{else}\ P_2
\end{aligned}$$

Syntactic transformations (2)

Single assignment renaming: when a variable is assigned at several places, rename it with a distinct name for each assignment.
(Not completely trivial because of array references.)

Example

$$\begin{array}{l}
\mathit{start}(); \textbf{new } r_A : T_r; \textbf{let } k_A = \mathit{kgen}(r_A) \textbf{ in} \\
\qquad \textbf{new } r_B : T_r; \textbf{let } k_B = \mathit{kgen}(r_B) \textbf{ in } \overline{c}\langle\rangle; (Q_K \mid Q_S) \\
Q_K = !^{i \leq n} c(h : T_h, k : T_k) \\
\qquad \textbf{if } h = A \textbf{ then let } k'_1 = k_A \textbf{ else} \\
\qquad \textbf{if } h = B \textbf{ then let } k'_2 = k_B \textbf{ else let } k'_3 = k \\
Q_S = !^{i' \leq n'} c'(h' : T_h); \\
\qquad \textbf{find } j \leq n \textbf{ suchthat defined}(h[j], k'_1[j]) \wedge h' = h[j] \textbf{ then } P_1(k'_1[j]) \\
\qquad \textbf{orfind } j \leq n \textbf{ suchthat defined}(h[j], k'_2[j]) \wedge h' = h[j] \textbf{ then } P_1(k'_2[j]) \\
\qquad \textbf{orfind } j \leq n \textbf{ suchthat defined}(h[j], k'_3[j]) \wedge h' = h[j] \textbf{ then } P_1(k'_3[j]) \\
\qquad \textbf{else } P_2
\end{array}$$

Syntactic transformations (3)

Move new: move restrictions downwards in the game as much as possible, when there is no array reference to them.
(Moving **new** $x : T$ under a **if** or a **find** duplicates it.
A subsequent single assignment renaming will distinguish cases.)

Example

$$\textbf{new}\ x : \mathit{nonce}; \textbf{if}\ c\ \textbf{then}\ P_1\ \textbf{else}\ P_2$$

becomes

$$\textbf{if}\ c\ \textbf{then}\ \textbf{new}\ x : \mathit{nonce}; P_1\ \textbf{else}\ \textbf{new}\ x : \mathit{nonce}; P_2$$

Syntactic transformations (4)

- Merge arrays: merge several variables $x_1, \ldots, x_n$ into a single variable x_1 when they are used for different indices (defined in different branches of a test **if** or **find**).
- Merge branches of **if** or **find** when they execute the same code, up to renaming of variables with array accesses.

Syntactic transformations (5): manual transformations

Insert an instruction: insert a test to distinguish cases; insert a variable definition; ...
Preserves the semantics of the game (e.g., the rest of the code is copied in both branches of the inserted test).

Example

P becomes

$$\textbf{if } \mathit{cond} \textbf{ then } P \textbf{ else } P$$

Subsequent transformations can transform P differently, depending on whether *cond* holds.

Syntactic transformations (6): manual transformations

- Insert an event: to apply Shoup's lemma.
  - A subprocess P becomes **event** e.
  - The probability of distinguishing the two games is the probability of executing event e. It will be bound by a proof by sequences of games.
- Replace a term with an equal term. CryptoVerif verifies that the terms are really equal.

Simplification and elimination of collisions

- CryptoVerif collects equalities that come from:
  - Assignments: **let** $x = M$ **in** P implies that $x = M$ in P
  - Tests: **if** $M = N$ **then** P implies that $M = N$ in P
  - Definitions of cryptographic primitives
  - When a **find** guarantees that $x[j]$ is defined, equalities that hold at definition of x also hold under the find (after substituting j for the array indices at the definition of x)
  - Elimination of collisions: if x is created by **new** $x : T$, $x[i] = x[j]$ implies $i = j$, up to negligible probability (when T is large)
- These equalities are combined to simplify terms.
- When terms can be simplified, processes are simplified accordingly. For instance:
  - If M simplifies to **true**, then **if** M **then** P_1 **else** P_2 simplifies P_1.
  - If a condition of **find** simplifies to **false**, then the corresponding branch is removed.

Proof of security properties: one-session secrecy

One-session secrecy: the adversary cannot distinguish any of the secrets from a random number with one test query.

Definition (One-session secrecy)

Assume that the variable x of type T is defined in G under a single $!^{i \leq n}$.

G preserves the one-session secrecy of x up to probability p when, for all evaluation contexts C acceptable for $G \mid Q_x$ with no public variables that do not contain S, $2\Pr[C[G \mid Q_x] \rightsquigarrow D_\mathsf{S}] - 1 \leq p(C)$ where

$$\begin{aligned} Q_x = c_0(); \textbf{new } b : \mathit{bool}; \overline{c_0}\langle\rangle;\ & (c(j : [1,n]); \textbf{if defined}(x[j]) \textbf{ then} \\ & \quad \textbf{if } b \textbf{ then } \overline{c}\langle x[j]\rangle \textbf{ else new } y : T; \overline{c}\langle y\rangle \\ & \mid c'(b' : \mathit{bool}); \textbf{if } b = b' \textbf{ then event } \mathsf{S}) \end{aligned}$$

$D_\mathsf{S}(\mathcal{E}) = (\mathsf{S} \in \mathcal{E})$, c_0, c, c', b, b', j, y, and S do not occur in G.

Proof of security properties: one-session secrecy

One-session secrecy: the adversary cannot distinguish any of the secrets from a random number with one test query.

Criterion for proving one-session secrecy of x:
x is defined by **new** $x[i] : T$ and there is a set of variables S such that only variables in S depend on x.
The output messages and the control-flow do not depend on x.

Proof of security properties: secrecy

Secrecy: the adversary cannot distinguish the secrets from independent random numbers with several test queries.

Criterion for proving secrecy of x: same as one-session secrecy, plus $x[i]$ and $x[i']$ do not come from the same copy of the same restriction when $i \neq i'$.

Proof strategy: advice

- One tries to execute each transformation given by the definition of a cryptographic primitive.
- When it fails, it tries to analyze why the transformation failed, and suggests syntactic transformations that could make it work.
- One tries to execute these syntactic transformations. (If they fail, they may also suggest other syntactic transformations, which are then executed.)
- We retry the cryptographic transformation, and so on.

Proof of the example: initial game

$$
\begin{aligned}
Q_0 = {} & start(); \textbf{new}\ r : keyseed; \textbf{let}\ k : key = kgen(r)\ \textbf{in} \\
& \textbf{new}\ r' : mkeyseed; \textbf{let}\ mk : mkey = mkgen(r')\ \textbf{in}\ \overline{c}\langle\rangle; (Q_A \mid Q_B)
\end{aligned}
$$

$$
\begin{aligned}
Q_A = {} & !^{i \le n} c_A(); \textbf{new}\ k' : key; \textbf{new}\ r'' : coins; \\
& \textbf{let}\ m : bitstring = enc(k2b(k'), k, r'')\ \textbf{in} \\
& \overline{c_A}\langle m, mac(m, mk)\rangle
\end{aligned}
$$

$$
\begin{aligned}
Q_B = {} & !^{i' \le n} c_B(m' : bitstring, ma : macstring); \\
& \textbf{if}\ verify(m', mk, ma)\ \textbf{then} \\
& \textbf{let}\ i_\perp(k2b(k'')) = dec(m', k)\ \textbf{in}\ \overline{c_B}\langle\rangle
\end{aligned}
$$

Proof of the example: remove assignments *mk*

$$Q_0 = start(); \textbf{new } r : keyseed; \textbf{let } k : key = kgen(r) \textbf{ in}$$
$$\textbf{new } r' : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$Q_A = !^{i \leq n} c_A(); \textbf{new } k' : key; \textbf{new } r'' : coins;$$
$$\textbf{let } m : bitstring = enc(k2b(k'), k, r'') \textbf{ in}$$
$$\overline{c_A}\langle m, mac(m, mkgen(r'))\rangle$$

$$Q_B = !^{i' \leq n} c_B(m' : bitstring, ma : macstring);$$
$$\textbf{if } verify(m', mkgen(r'), ma) \textbf{ then}$$
$$\textbf{let } i_\perp(k2b(k'')) = dec(m', k) \textbf{ in } \overline{c_B}\langle\rangle$$

Proof of the example: security of the MAC

$$Q_0 = start(); \textbf{new } r : keyseed; \textbf{let } k : key = kgen(r) \textbf{ in}$$
$$\textbf{new } r' : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$Q_A = !^{i \leq n} c_A(); \textbf{new } k' : key; \textbf{new } r'' : coins;$$
$$\textbf{let } m : bitstring = enc(k2b(k'), k, r'') \textbf{ in}$$
$$\overline{c_A}\langle m, mac'(m, mkgen'(r'))\rangle$$

$$Q_B = !^{i' \leq n} c_B(m' : bitstring, ma : macstring);$$
$$\textbf{find } j \leq n \textbf{ suchthat defined}(m[j]) \wedge m' = m[j] \wedge$$
$$verify'(m', mkgen'(r'), ma) \textbf{ then}$$
$$\textbf{let } i_\perp(k2b(k'')) = dec(m', k) \textbf{ in } \overline{c_B}\langle\rangle$$

Probability: $\mathrm{Succ}^{\mathrm{uf-cma}}_{\mathrm{MAC}}(\textbf{time} + \textbf{time}(kgen) + n\,\textbf{time}(enc, \mathrm{length}(key)) + n\,\textbf{time}(dec, \mathrm{maxl}(m')), n, n, \max(\mathrm{maxl}(m'), \mathrm{maxl}(m)))$.

Proof of the example: simplify

$$Q_0 = start(); \textbf{new } r : keyseed; \textbf{let } k : key = kgen(r) \textbf{ in}$$
$$\textbf{new } r' : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$Q_A = !^{i \leq n} c_A(); \textbf{new } k' : key; \textbf{new } r'' : coins;$$
$$\textbf{let } m : bitstring = enc(k2b(k'), k, r'') \textbf{ in}$$
$$\overline{c_A}\langle m, mac'(m, mkgen'(r'))\rangle$$

$$Q_B = !^{i' \leq n} c_B(m' : bitstring, ma : macstring);$$
$$\textbf{find } j \leq n \textbf{ suchthat defined}(m[j]) \wedge m' = m[j] \wedge$$
$$verify'(m', mkgen'(r'), ma) \textbf{ then}$$
$$\textbf{let } {\color{red}k''} = {\color{red}k'[j]} \textbf{ in } \overline{c_B}\langle\rangle$$

$$dec(m', k) = dec(enc(k2b(k'[j]), k, r''[j]), k) = i_\perp(k2b(k'[j]))$$

Proof of the example: remove assignments k

$$Q_0 = start(); \textbf{new } r : keyseed; \textbf{new } r' : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$\begin{aligned}
Q_A = {}&!^{i \leq n} c_A(); \textbf{new } k' : key; \textbf{new } r'' : coins;\\
&\textbf{let } m : bitstring = enc(k2b(k'), kgen(r), r'') \textbf{ in}\\
&\overline{c_A}\langle m, mac'(m, mkgen'(r'))\rangle
\end{aligned}$$

$$\begin{aligned}
Q_B = {}&!^{i' \leq n} c_B(m' : bitstring, ma : macstring);\\
&\textbf{find } j \leq n \textbf{ suchthat defined}(m[j]) \wedge m' = m[j] \wedge\\
&\quad verify'(m', mkgen'(r'), ma) \textbf{ then}\\
&\textbf{let } k'' = k'[j] \textbf{ in } \overline{c_B}\langle\rangle
\end{aligned}$$

Proof of the example: security of the encryption

$$Q_0 = start(); \textbf{new } r : keyseed; \textbf{new } r' : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$\begin{aligned}Q_A = &!^{i \leq n} c_A(); \textbf{new } k' : key; \textbf{new } r'' : coins;\\ &\textbf{let } m : bitstring = enc'(Z(k2b(k')), kgen'(r), r'') \textbf{ in}\\ &\overline{c_A}\langle m, mac'(m, mkgen'(r'))\rangle\end{aligned}$$

$$\begin{aligned}Q_B = &!^{i' \leq n} c_B(m' : bitstring, ma : macstring);\\ &\textbf{find } j \leq n \textbf{ suchthat defined}(m[j]) \wedge m' = m[j] \wedge\\ &\quad verify'(m', mkgen'(r'), ma) \textbf{ then}\\ &\textbf{let } k'' = k'[j] \textbf{ in } \overline{c_B}\langle\rangle\end{aligned}$$

Probability: $\mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(\textbf{time} + (n + n^2)\textbf{time}(mkgen) + n\,\textbf{time}(mac, \mathrm{maxl}(m)) + n^2\,\textbf{time}(verify, \mathrm{maxl}(m')) + n^2\,\textbf{time}(= bitstring, \mathrm{maxl}(m'), \mathrm{maxl}(m)), n, \mathrm{length}(key))$

Proof of the example: security of the encryption

$$Q_0 = start(); \textbf{new } r : keyseed; \textbf{new } r' : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$\begin{aligned}Q_A = {}& !^{i \leq n} c_A(); \textbf{new } k' : key; \textbf{new } r'' : coins;\\ & \textbf{let } m : bitstring = enc'(Z(k2b(k')), kgen'(r), r'') \textbf{ in}\\ & \overline{c_A}\langle m, mac'(m, mkgen'(r'))\rangle\end{aligned}$$

$$\begin{aligned}Q_B = {}& !^{i' \leq n} c_B(m' : bitstring, ma : macstring);\\ & \textbf{find } j \leq n \textbf{ suchthat defined}(m[j]) \wedge m' = m[j] \wedge\\ & \quad verify'(m', mkgen'(r'), ma) \textbf{ then}\\ & \textbf{let } k'' = k'[j] \textbf{ in } \overline{c_B}\langle\rangle\end{aligned}$$

Better probability: $\mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(\textbf{time} + (n + n^2)\textbf{time}(mkgen) + n\,\textbf{time}(mac, \mathrm{maxl}(m)) + n^2\,\textbf{time}(verify, \mathrm{maxl}(m')) + n^2\,\textbf{time}(= bitstring, \mathrm{maxl}(m'), \mathrm{maxl}(m)), n, \mathrm{length}(key))$

Proof of the example: simplify

$$Q_0 = start(); \textbf{new } r : keyseed; \textbf{new } r' : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$\begin{aligned}Q_A = &!^{i \leq n} c_A(); \textbf{new } k' : key; \textbf{new } r'' : coins;\\ &\textbf{let } m : bitstring = enc'(Z_k, kgen'(r), r'') \textbf{ in}\\ &\overline{c_A}\langle m, mac'(m, mkgen'(r'))\rangle\end{aligned}$$

$$\begin{aligned}Q_B = &!^{i' \leq n} c_B(m' : bitstring, ma : macstring);\\ &\textbf{find } j \leq n \textbf{ suchthat defined}(m[j]) \wedge m' = m[j] \wedge\\ &\quad verify'(m', mkgen'(r'), ma) \textbf{ then}\\ &\textbf{let } k'' = k'[j] \textbf{ in } \overline{c_B}\langle\rangle\end{aligned}$$

$$Z(k2b(k')) = Z_k$$

Proof of the example: secrecy

$$Q_0 = start(); \textbf{new } r : keyseed; \textbf{new } r' : mkeyseed; \overline{c}\langle\rangle; (Q_A \mid Q_B)$$

$$\begin{aligned} Q_A = !^{i \leq n} c_A(); & \textbf{new } k' : key; \textbf{new } r'' : coins; \\ & \textbf{let } m : bitstring = enc'(Z_k, kgen'(r), r'') \textbf{ in} \\ & \overline{c_A}\langle m, mac'(m, mkgen'(r'))\rangle \end{aligned}$$

$$\begin{aligned} Q_B = !^{i' \leq n} c_B(& m' : bitstring, ma : macstring); \\ & \textbf{find } j \leq n \textbf{ suchthat defined}(m[j]) \wedge m' = m[j] \wedge \\ & \quad verify'(m', mkgen'(r'), ma) \textbf{ then} \\ & \textbf{let } k'' = k'[j] \textbf{ in } \overline{c_B}\langle\rangle \end{aligned}$$

Preserves the one-session secrecy of k'' but not its secrecy.

Final result

Adding the probabilities, we obtain:

Result

The probability that an adversary that runs in time at most t, that executes n sessions of A and B and sends messages of length at most l_{mB} to B breaks the one-session secrecy of k'' is

$$2\mathrm{Succ}_{\mathrm{MAC}}^{\mathrm{uf-cma}}(t'_1, n, n, \max(l_{mB}, l_c)) + 2\mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(t'_2, n, l_k)$$

where $t'_1 = t + \mathbf{time}(kgen) + n\,\mathbf{time}(enc, l_k) + n\,\mathbf{time}(dec, l_{mB})$

$t'_2 = t + (n + n^2)\mathbf{time}(mkgen) + n\,\mathbf{time}(mac, l_c) + n^2\,\mathbf{time}(verify, l_{mB}) + n^2\,\mathbf{time}(= bitstring, l_{mB}, l_c)$

l_k is the length of keys, l_c the length of encryptions of keys.

The factor 2 comes from the definition of secrecy.

Exercise 5: preliminary definition

Definition (IND-CCA2 symmetric encryption)

A symmetric encryption scheme SE is indistinguishable under adaptive chosen-ciphertext attacks (IND-CCA2) if and only if $\mathrm{Succ}^{\mathrm{ind-cca2}}_{\mathrm{SE}}(t, q_e, q_d, l_e, l_d)$ is negligible when t is polynomial in the security parameter:

$$\mathrm{Succ}^{\mathrm{ind-cca2}}_{\mathrm{SE}}(t, q_e, q_d, l_e, l_d) = \max_{\mathcal{A}} 2 \Pr\left[\begin{array}{l} b \stackrel{R}{\leftarrow} \{0,1\}; k \stackrel{R}{\leftarrow} kgen; \\ b' \leftarrow \mathcal{A}^{enc(LR(.,.,b),k),dec(.,k)} : b' = b \wedge \\ \mathcal{A} \text{ has not called } dec(.,k) \text{ on the result of} \\ enc(LR(.,.,b),k) \end{array}\right] - 1$$

where $\mathcal{A}$ runs in time at most t,
calls $enc(LR(.,.,b),k)$ at most q_e times on messages of length at most l_e,
calls $dec(.,k)$ at most q_d times on messages of length at most l_d.

Exercise 5

1. Show using CryptoVerif that, if the MAC scheme is SUF-CMA and the encryption scheme is IND-CPA, then the encrypt-then-MAC scheme is IND-CCA2.
2. Show using the same assumptions that the encrypt-then-MAC scheme is INT-CTXT.
3. What happens if the MAC scheme is only UF-CMA?

Example of the FDH signature (joint work with D. Pointcheval)

hash hash function (in the random oracle model)
$\mathrm{f}(pk, m)$ one-way trapdoor permutation, with inverse $\mathrm{invf}(sk, m)$.

We define a signature scheme as follows:

- signature $\mathrm{sign}(m, sk) = \mathrm{invf}(sk, \mathrm{hash}(hk, m))$
- verification $\mathrm{verify}(m, pk, s) = (\mathrm{f}(pk, s) = \mathrm{hash}(hk, m))$

Our goal is to show that this signature scheme is UF-CMA (secure against existential forgery under chosen message attacks).

Formalizing the security of a signature scheme (1)

Key generation:

$start(); \mathbf{new}\ r : keyseed; \mathbf{let}\ pk = \mathrm{pkgen}(r)\ \mathbf{in}\ \mathbf{let}\ sk = \mathrm{skgen}(r)\ \mathbf{in}\ \overline{c0}\langle pk\rangle$

Chooses a random seed uniformly in the set of bit-strings *keyseed* (consisting of all bit-strings of a certain length), generates a public key *pk*, a secret key *sk*, and outputs the public key.

Formalizing the security of a signature scheme (2)

Signature:

$$c1(m : \mathit{bitstring}); \overline{c2}\langle \mathrm{sign}(sk, m)\rangle$$

Formalizing the security of a signature scheme (2)

Signature:

$$c1(m : bitstring); \overline{c2}\langle \mathrm{sign}(sk, m)\rangle$$

This process can be called at most q_S times:

$$!^{i_S \leq q_S} c1(m : bitstring); \overline{c2}\langle \mathrm{sign}(sk, m)\rangle$$

Formalizing the security of a signature scheme (2)

Signature:

$$c1(m : \mathit{bitstring}); \overline{c2}\langle \mathrm{sign}(sk, m)\rangle$$

This process can be called at most q_S times:

$$!^{i_S \leq q_S} c1(m : \mathit{bitstring}); \overline{c2}\langle \mathrm{sign}(sk, m)\rangle$$

In fact, this is an abbreviation for:

$$!^{i_S \leq q_S} c1(m[i_S] : \mathit{bitstring}); \overline{c2}\langle \mathrm{sign}(sk, m[i_S])\rangle$$

The variables in repeated oracles are arrays, with one cell for each call, to remember the values used in each oracle call.
These arrays are indexed with the call number i_S.

Formalizing the security of a signature scheme (3)

Test:

$$c3(m' : \mathit{bitstring}, s : D); \textbf{if } \mathrm{verify}(m', pk, s) \textbf{ then}$$
$$\textbf{find } j \leq q_S \textbf{ suchthat defined}(m[j]) \wedge (m' = m[j])$$
$$\textbf{then yield else event } \mathrm{bad})$$

If s is a signature for m' and the signed message m' is not contained in the array m of messages passed to signing oracle, then the signature is a forgery, so we execute **event** bad.

Formalizing the security of a signature scheme (summary)

The signature and test oracles make sense only after the key generation oracle has been called, hence a sequential composition.

The signature and test oracles are simultaneously available, hence a parallel composition.

$start(); \textbf{new}\ r : keyseed; \textbf{let}\ pk = \mathrm{pkgen}(r)\ \textbf{in let}\ sk = \mathrm{skgen}(r)\ \textbf{in}\ \overline{c0}\langle pk\rangle;$

$((\text{* signature oracle *})$

$\quad !^{is \leq qs} c1(m : bitstring); \overline{c2}\langle \mathrm{sign}(sk, m)\rangle$

$\mid (\text{* forged signature? *})$

$\quad c3(m' : bitstring, s : D); \textbf{if}\ \mathrm{verify}(m', pk, s)\ \textbf{then}$

$\quad \textbf{find}\ j \leq q_S\ \textbf{suchthat defined}(m[j]) \wedge (m' = m[j])$

$\quad \textbf{then yield else event}\ \mathrm{bad})$

The probability of executing **event** bad in this game is the probability of forging a signature.

Application to the FDH signature scheme

We add a hash oracle because the adversary must be able to call the random oracle (even though it cannot be implemented).

$$
\begin{array}{l}
\mathit{start}();\ \textbf{new}\ \mathit{hk} : \mathit{hashkey}; \textbf{new}\ r : \mathit{keyseed};\\
\textbf{let}\ \mathit{sk} = \mathrm{skgen}(r)\ \textbf{in let}\ \mathit{pk} = \mathrm{pkgen}(r)\ \textbf{in}\ \overline{c0}\langle \mathit{pk}\rangle;\\
((\text{(* hash oracle *)})\ !^{i_H \leq q_H} \mathit{hc}1(x : \mathit{bitstring});\ \overline{\mathit{hc}2}\langle \mathrm{hash}(\mathit{hk}, x)\rangle\\
\mid \text{(* signature oracle *)}\ !^{i_S \leq q_S} c1(m : \mathit{bitstring});\ \overline{c2}\langle \mathrm{invf}(\mathit{sk}, \mathrm{hash}(\mathit{hk}, m))\rangle\\
\mid \text{(* forged signature? *)}\\
\quad c3(m' : \mathit{bitstring}, s : D); \textbf{if}\ \mathrm{f}(\mathit{pk}, s) = \mathrm{hash}(\mathit{hk}, m')\ \textbf{then}\\
\quad \textbf{find}\ j \leq q_S\ \textbf{suchthat defined}(m[j]) \wedge (m' = m[j])\\
\quad \textbf{then yield else event}\ \mathrm{bad})
\end{array}
$$

Our goal is to bound the probability that **event** bad is executed in this game.
This game is given as input to the prover in the syntax above.

FDH: security of a hash function

A hash function is equivalent to a "random function": a function that

- returns a new random number when it is a called on a new argument,
- and returns the same result when it is called on the same argument.

$$
\begin{array}{l}
!^{Nh}\ \textbf{new}\ k : \mathit{hashkey}; !^{N} \mathit{Ohash}(x : \mathit{bitstring}) := \mathrm{hash}(k, x)\\
\approx_0\\
!^{Nh}\ \textbf{new}\ k : \mathit{hashkey}; !^{N} \mathit{Ohash}(x : \mathit{bitstring}) :=\\
\quad \textbf{find}\ j \leq N\ \textbf{suchthat}\ \textbf{defined}(x[j], r[j])\ \&\&\ (x = x[j])\\
\quad \textbf{then}\ r[j]\\
\quad \textbf{else}\ \textbf{new}\ r : D;\ r
\end{array}
$$

FDH: security of a hash function (optimized)

For a test $r' = h(x')$, we can avoid computing $h(x')$ explicitly:

- if x' has been passed to the hash function previously, compare r' with the previous result;
- otherwise, return false.

In the latter case, test indeed false, except when the fresh random number $h(x')$ collides with r' (probability $1/|D|$).

$$
\begin{array}{l}
!^{Nh}\ \textbf{new}\ k : \mathit{hashkey};\\
\quad (!^{N} \mathit{Ohash}(x : \mathit{bitstring}) := \mathrm{hash}(k, x),\\
\quad\ !^{Neq} \mathit{Oeq}(x' : \mathit{bitstring}; r' : D) := r' = \mathrm{hash}(k, x'))
\end{array}
$$

$$\approx_{\#Oeq/|D|}$$

$$
\begin{array}{l}
!^{Nh}(!^{N} \mathit{Ohash}(x : \mathit{bitstring}) := \textbf{find}\ j \leq N\ \textbf{suchthat}\\
\qquad \textbf{defined}(x[j], r[j])\ \&\&\ (x = x[j])\ \textbf{then}\ r[j]\ \textbf{else new}\ r : D;\ \ r,\\
\quad\ !^{Neq} \mathit{Oeq}(x' : \mathit{bitstring}; r' : D) := \textbf{find}\ j \leq N\ \textbf{suchthat}\\
\qquad \textbf{defined}(x[j], r[j])\ \&\&\ (x' = x[j])\ \textbf{then}\ r' = r[j]\ \textbf{else false})
\end{array}
$$

FDH: one-wayness

The adversary inverts f when, given the public key $pk = \mathrm{pkgen}(r)$ and the image of some x by $\mathrm{f}(pk, \cdot)$, it manages to find x (without having the trapdoor).

The function f is one-way when the adversary has negligible probability of inverting f.

Definition (One-wayness)

$$\mathrm{Succ}^{\mathrm{ow}}_{\mathcal{P}}(t) = \max_{\mathcal{A}} \Pr\left[\begin{array}{l} r \overset{R}{\leftarrow} keyseed, pk \leftarrow \mathrm{pkgen}(r), x \overset{R}{\leftarrow} D, \\ y \leftarrow \mathrm{f}(pk, x), x' \leftarrow \mathcal{A}(pk, y) : x = x' \end{array}\right]$$

where $\mathcal{A}$ runs in time at most t.

FDH: one-wayness

$$
\begin{array}{l}
!^{Nk}\ \textbf{new}\ r : \mathit{keyseed};\ (\\
\quad \mathit{Opk}() := \mathrm{pkgen}(r), \\
\quad !^{Nf}\ \textbf{new}\ x : D;\ (\\
\qquad \mathit{Oy}() := \mathrm{f}(\mathrm{pkgen}(r), x), \\
\qquad !^{N2} \mathit{Oeq}(x' : D) := (x' = x), \\
\qquad \mathit{Ox}() := x)) \\
\approx_{Nk \times Nf \times \mathrm{Succ}^{\mathrm{ow}}_{\mathcal{P}}(\textbf{time} + (Nk-1)\times \textbf{time}(\mathrm{pkgen}) + (\#\mathit{Oy}-1)\times \textbf{time}(\mathrm{f}))} \\
!^{Nk}\ \textbf{new}\ r : \mathit{keyseed};\ (\\
\quad \mathit{Opk}() := \mathrm{pkgen}'(r), \\
\quad !^{Nf}\ \textbf{new}\ x : D;\ (\\
\qquad \mathit{Oy}() := \mathrm{f}'(\mathrm{pkgen}'(r), x), \\
\qquad !^{N2} \mathit{Oeq}(x' : D) := \textbf{if}\ \textbf{defined}(k)\ \textbf{then}\ x' = x\ \textbf{else}\ \mathrm{false}, \\
\qquad \mathit{Ox}() := \textbf{let}\ k : \mathit{bitstring} = \mathrm{mark}\ \textbf{in}\ x))
\end{array}
$$

FDH: other properties of one-way trapdoor permutations

invf is the inverse of f:

$$\forall r : keyseed, x : D; \mathrm{invf}(\mathrm{skgen}(r), \mathrm{f}(\mathrm{pkgen}(r), x)) = x$$

f is injective:

$$\forall k : key, x : D, x' : D; (\mathrm{f}(k, x) = \mathrm{f}(k, x')) = (x = x')$$

We can replace a uniformly distributed random number y with $\mathrm{f}(\mathrm{pkgen}(r), y')$ where y' is a uniformly distributed random number:

$!^{Nk}$ **new** $r : keyseed;($
  $Opk() := \mathrm{pkgen}(r),$
  $!^{Nf}$ **new** $y : D; (Oant() := \mathrm{invf}(\mathrm{skgen}(r), y), Oim() := y))$

$\approx_0$

$!^{Nk}$ **new** $r : keyseed;($
  $Opk() := \mathrm{pkgen}(r),$
  $!^{Nf}$ **new** $x : D; (Oant() := x, Oim() := \mathrm{f}(\mathrm{pkgen}(r), y)))$

Demo

- CryptoVerif input file: examples/fdh
- library of primitives
- run CryptoVerif
- output

FDH: initial game

$start()$; **new** $hk : hashkey$; **new** $r : keyseed$;
let $sk : key = \mathrm{skgen}(r)$ **in**
let $pk : key = \mathrm{pkgen}(r)$ **in** $\overline{c0}\langle pk\rangle$;
((* hash oracle *)
  $!^{i_H \leq q_H} hc1[i_H](x : bitstring)$; $\overline{hc2[i_H]}\langle \mathrm{hash}(hk, x)\rangle$
| (* signature oracle *)
  $!^{i_S \leq q_S} c1[i_S](m : bitstring)$; $\overline{c2[i_S]}\langle \mathrm{invf}(sk, \mathrm{hash}(hk, m))\rangle$
| (* forged signature? *)
  $c3(m' : bitstring, s : D)$;
  if $\mathrm{f}(pk, s) = \mathrm{hash}(hk, m')$ **then**
  find $j \leq q_S$ **suchthat defined**$(m[j])$ && $(m' = m[j])$ **then**
    yield
  else
    event bad
)

FDH step 1: apply the security of the hash function

Replace each occurrence of hash(M) with a lookup in the arguments of previous calls to hash.

- If M is found, return the same result as the previous result.
- Otherwise, pick a new random number and return it.

For instance, $\overline{hc2[i_H]}\langle \mathrm{hash}(hk, x)\rangle$ is replaced with

$\textbf{find } @i1 \leq q_S \textbf{ suchthat defined}(m[@i1], r\_32[@i1])$
$\quad \&\& \ (x = m[@i1]) \textbf{ then } \overline{hc2[i_H]}\langle r\_32[@i1]\rangle$
$\textbf{orfind } @i2 \leq q_H \textbf{ suchthat defined}(x[@i2], r\_34[@i2])$
$\quad \&\& \ (x = x[@i2]) \textbf{ then } \overline{hc2[i_H]}\langle r\_34[@i2]\rangle$
$\textbf{else}$
$\quad \textbf{new } r\_34 : D; \ \overline{hc2[i_H]}\langle r\_34\rangle$

The test $\mathrm{f}(pk, s) = \mathrm{hash}(hk, m')$ uses Oeq. Probability difference $1/|D|$.

FDH step 2: simplify

(* forged signature? *)
$c3(m' : bitstring, s : D);$
find $@i5 \leq q_S$ **suchthat defined**$(m[@i5], r\_32[@i5])$ && $(m' = m[@i5])$ **then**
  if $(\mathsf{f}(pk, s) = r\_32[@i5])$ **then**
  find $j \leq q_S$ **suchthat defined**$(m[j])$ && $(m' = m[j])$ **then yield else event** bad
orfind $@i6 \leq q_H$ **suchthat defined**$(x[@i6], r\_34[@i6])$ && $(m' = x[@i6])$ **then**
  if $(\mathsf{f}(pk, s) = r\_34[@i6])$ **then**
  find $j \leq q_S$ **suchthat defined**$(m[j])$ && $(m' = m[j])$ **then yield else event** bad
else
  if false then
  find $j \leq q_S$ **suchthat defined**$(m[j])$ && $(m' = m[j])$ **then yield else event** bad

The red test always succeeds, so the blue part becomes **yield**.
The magenta part becomes **yield**.

FDH step 3: substitute *sk* with its value

The variable sk is replaced with $\mathrm{skgen}(r)$, and the assignment **let** $sk : key = \mathrm{skgen}(r)$ is removed.
This transformation is advised in order to able to apply the permutation property.

FDH step 4: permutation

(* signature oracle *)
$!^{i_S \leq q_S}$
$c1[i_S](m : bitstring);$
find $@i3 \leq q_S$ **suchthat defined**$(m[@i3], r\_32[@i3])$ && $(m = m[@i3])$ **then**
$\overline{c2[i_S]}\langle \mathrm{invf}(\mathrm{skgen}(r), r\_32[@i3])\rangle$
orfind $@i4 \leq q_H$ **suchthat defined**$(x[@i4], r\_34[@i4])$ && $(m = x[@i4])$ **then**
$\overline{c2[i_S]}\langle \mathrm{invf}(\mathrm{skgen}(r), r\_34[@i4])\rangle$
else
new $r\_32 : D$;
$\overline{c2[i_S]}\langle \mathrm{invf}(\mathrm{skgen}(r), r\_32)\rangle$

new $r\_i : D$ becomes **new** $y\_i : D$,
$\mathrm{invf}(\mathrm{skgen}(r), r\_i)$ becomes y_i,
r_i becomes $\mathrm{f}(\mathrm{pkgen}(r), y\_i)$

FDH step 5: simplify

(* forged signature? *)
$c3(m' : bitstring, s : D);$
find $@i5 \leq q_S$ **suchthat defined**$(m[@i5], r\_32[@i5])$ && $(m' = m[@i5])$ **then**
  yield
orfind $@i6 \leq q_H$ **suchthat defined**$(x[@i6], r\_34[@i6])$ && $(m' = x[@i6])$ **then**
  if $(\mathrm{f}(pk, s) = \mathrm{f}(\mathrm{pkgen}(r), y\_34[@i6]))$ **then**
  find $j \leq q_S$ **suchthat defined**$(m[j])$ && $(m' = m[j])$ **then yield else event** bad

$\mathrm{f}(pk, s) = \mathrm{f}(\mathrm{pkgen}(r), y\_i)$ becomes $s = y\_i$,
knowing $pk = pkgen(r)$ and the injectivity of f:
$\forall k : key, x : D, x' : D; (\mathrm{f}(k, x) = \mathrm{f}(k, x')) = (x = x')$

FDH step 6: one-wayness

(* forged signature? *)
$c3(m' : bitstring, s : D);$
find $@i5 \leq q_S$ **suchthat defined**$(m[@i5], r\_32[@i5])$ && $(m' = m[@i5])$ **then**
yield
orfind $@i6 \leq q_H$ **suchthat defined**$(x[@i6], r\_34[@i6])$ && $(m' = x[@i6])$ **then**
if $s = y\_34[@i6]$ **then**
find $j \leq q_S$ **suchthat defined**$(m[j])$ && $(m' = m[j])$ **then yield else event** bad

$s = y\_i$ becomes **find** $@j\_i \leq q_H$ **suchthat defined**$(k\_i[@j\_i])$
then $s = y\_i$ **else** false,
In hash oracle, $\mathrm{f}(\mathrm{pkgen}(r), y\_i)$ becomes $\mathrm{f}'(\mathrm{pkgen}'(r), y\_i)$,
In signature oracle, y_i becomes **let** $k\_i : bitstring = \mathrm{mark}$ **in** y_i.
Difference of probability: $(q_H + q_S)\mathrm{Succ}_{\mathcal{P}}^{\mathrm{ow}}(\mathbf{time} + (q_H - 1)\mathbf{time}(f))$.

FDH step 7: simplify

(* forged signature? *)
$c3(m' : bitstring, s : D)$;
find $@i5 \leq q_S$ **suchthat defined**$(m[@i5], r\_32[@i5])$ && $(m' = m[@i5])$ **then**
  yield
orfind $@i6 \leq q_H$ **suchthat defined**$(x[@i6], r\_34[@i6])$ && $(m' = x[@i6])$ **then**
  find $@j\_34 \leq q_S$ **suchthat defined**$(k\_34[@j\_34])$ && $(@i4[@j\_34] = @i6)$ **then**
  if $s = y\_34[@i6]$ **then**
  find $j \leq q_S$ **suchthat defined**$(m[j])$ && $(m' = m[j])$ **then yield else event** bad

The test in red always succeeds, so **event** bad disappears, which proves the desired property.

FDH step 7: simplify (2)

(* forged signature? *)
$c3(m' : \mathit{bitstring}, s : D);$
$\ldots$
orfind $@i6 \leq q_H$ **suchthat defined**$(x[@i6], r\_34[@i6])$ && $(m' = x[@i6])$ **then**
  find $@j\_34 \leq q_S$ **suchthat defined**$(k\_34[@j\_34])$ && $(@i4[@j\_34] = @i6)$ **then**
  if $s = y\_34[@i6]$ **then**
  find $j \leq q_S$ **suchthat defined**$(m[j])$ && $(m' = m[j])$ **then yield else event** bad

Definition of k_34:

$!^{i_S \leq q_S}$
$c1[i_S](m : \mathit{bitstring});$
$\ldots$
orfind $@i4 \leq q_H$ **suchthat defined**$(x[@i4], y\_34[@i4])$ && $(m = x[@i4])$ **then**
  let $k\_34 : \mathit{bitstring} =$ mark **in** $\ldots$

When $k\_34[@j\_34]$ is defined, $m[@j\_34]$ is defined and
$m[@j\_34] = x[@i4[@j\_34]] = x[@i6] = m'$
so the red test succeeds with $j = @j\_34$.

FDH: final result

Adding the probabilities, we obtain:

Result

The probability that an adversary that runs in time at most t and makes q_S signature queries and q_H hash queries forges a FDH signature is at most

$$1/|D| + (q_S + q_H)\mathrm{Succ}^{\mathrm{ow}}_{\mathcal{P}}(t + (q_H - 1)\mathbf{time}(f))$$

Exercise 6

Suppose that H is a hash function in the Random Oracle Model and that f is a one-way trapdoor permutation.
Consider the encryption function $E_{pk}(x) = f_{pk}(r) || H(r) \oplus x$, where $||$ denotes concatenation and $\oplus$ denotes exclusive or (Bellare & Rogaway, CCS'93).

- What is the decryption function?
- Show using CryptoVerif that this public-key encryption scheme is IND-CPA. (IND-CPA is defined like IND-CCA2 except that the adversary does not have access to a decryption oracle.)

Experiments

Tested on the following protocols (original and corrected versions):

– Otway-Rees (shared-key)

– Yahalom (shared-key)

– Denning-Sacco (public-key)

– Woo-Lam shared-key and public-key

– Needham-Schroeder shared-key and public-key

Shared-key encryption is implemented as encrypt-then-MAC, using a IND-CPA encryption scheme.

(For Otway-Rees, we also considered a SPRP encryption scheme,
a IND-CPA + INT-CTXT encryption scheme,
a IND-CCA2 + IND-PTXT encryption scheme.)

Public-key encryption is assumed to be IND-CCA2.

We prove secrecy of session keys and correspondence properties.

Results (1)

In most cases, the prover succeeds in proving the desired properties when they hold, and obviously it always fails to prove them when they do not hold.

Only cases in which the prover fails although the property holds:

- Needham-Schroeder public-key when the exchanged key is the nonce N_A.
- Needham-Schroeder shared-key: fails to prove that $N_B[i] \neq N_B[i'] - 1$ with overwhelming probability, where N_B is a nonce
- Showing that the encryption scheme $\mathcal{E}(m, r) = f(r) \| H(r) \oplus m \| H'(m, r)$ is IND-CCA2.

Results (2)

- Some public-key protocols need manual proofs.
  (Give the cryptographic proof steps and single assignment renaming instructions.)
- Runtime: 7 ms to 35 s, average: 5 s on a Pentium M 1.8 GHz.

Other case studies

- Full domain hash signature (with David Pointcheval)
  Encryption schemes of Bellare-Rogaway'93 (with David Pointcheval)
- Kerberos V, with and without PKINIT (with Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay).
- OEKE (variant of Encrypted Key Exchange, with David Pointcheval).
- A part of an F# implementation of the TLS transport protocol (Microsoft Research and MSR-INRIA).

Conclusion

CryptoVerif can automatically prove the security of primitives and protocols.

- The security assumptions are given as observational equivalences (proved manually once).
- The protocol or scheme to prove is specified in a process calculus.
- The prover provides a sequence of indistinguishable games that lead to the proof and a bound on the probability of an attack.
- The user is allowed (but does not have) to interact with the prover to make it follow a specific sequence of games.

Future work: CryptoVerif extensions

- Support more primitives:
  - More equations, *e.g.* associativity for XOR
  - Primitives with internal state
- Improvements in the proof strategy.
  More precise manual hints?
- More case studies.
  - Will suggest more extensions.
- Certify CryptoVerif; combine it with CertiCrypt.

Future work: grand challenges

- Proof of implementations of protocols in the computational model:
  - by analysis of existing implementations,
  - by generation of implementations from specifications.
- Take into account side-channels.

Acknowledgments

- I warmly thank David Pointcheval for his advice and explanations of the computational proofs of protocols. This project would not have been possible without him.
- This work was partly supported by the ANR project FormaCrypt (ARA SSIA 2005).
- This work is partly supported by the ANR project ProSe (VERSO 2010).